Multi-level Analysis of GPU Utilization in ML Training Workloads

Paul Delestrac, Debjyoti Bhattacharjee, Simei Yang, Diksha Moolchandani, Francky Catthoor, Lionel Torres, David Novo

To cite this version:

Paul Delestrac, Debjyoti Bhattacharjee, Simei Yang, Diksha Moolchandani, Francky Catthoor, et al.. Multi-level Analysis of GPU Utilization in ML Training Workloads. DATE 2024 - 27th Design, Automation and Test in Europe Conference, Mar 2024, Valencia, Spain. hal-04523554

HAL Id: hal-04523554

https://hal.umontpellier.fr/hal-04523554v1

Submitted on 30 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Multi-level Analysis of GPU Utilization in ML Training Workloads

Paul Delestrac[†] Debjyoti Bhattacharjee[‡] Simei Yang[‡] Diksha Moolchandani[‡]
Francky Catthoor[‡*] Lionel Torres[†] David Novo[†]

[†]LIRMM, Univ. Montpellier, CNRS, Montpellier, France [‡]IMEC, Leuven, Belgium [*]KU Leuven, Leuven, Belgium

***Abstract*—Training time has become a critical bottleneck due to the recent proliferation of large-parameter ML models. GPUs continue to be the prevailing architecture for training ML models. However, the complex execution flow of ML frameworks makes it difficult to understand GPU computing resource utilization. Our main goal is to provide a better understanding of how efficiently ML training workloads use the computing resources of modern GPUs. To this end, we first describe an ideal reference execution of a GPU-accelerated ML training loop and identify relevant metrics that can be measured using existing profiling tools. Second, we produce a coherent integration of the traces obtained from each profiling tool. Third, we leverage the metrics within our integrated trace to analyze the impact of different software optimizations (e.g., mixed-precision, various ML frameworks, and execution modes) on the throughput and the associated utilization at multiple levels of hardware abstraction (i.e., whole GPU, SM subpartitions, issue slots, and tensor cores). In our results on two modern GPUs, we present seven takeaways and show that although close to 100% utilization is generally achieved at the GPU level, average utilization of the issue slots and tensor cores always remains below 50% and 5.2%, respectively.**

I. Introduction

Recent years have seen an explosion of new Deep Learning (DL) models (e.g., Convolutional Neural Networks (CNNs), transformer-based large language models (LLMs), etc.) with increasing amounts of trainable parameters (e.g., GPT-3 has 175 billion parameters [1]). With this increase in parameters, the training time is now a major bottleneck, pushing for additional compute power for model development.

Although extensive research is conducted in new specialized accelerator design [2], GPUs remain the prevailing architecture for training Machine Learning (ML) workloads. Previous works [3]–[5] show that GPU architectures have evolved (e.g., the addition of tensor cores in 2017) to increase throughput and reduce ML training latency. However, throughput and latency improvements can be achieved through brute force approaches, which do not necessarily translate to a well-balanced use of computing resources, wasting area/cost and inducing energy overheads. Thus, to design more compute-efficient accelerators, it is important to evaluate how efficiently ML training workloads use the computing resources of modern architectures such as GPUs.

However, state-of-the-art GPU architectures are intricate proprietary designs and their interaction with ML frameworks relies on complex runtimes [6] and optimized closed-source libraries [7]. This makes gathering performance metrics tedious as it requires using multiple profiling tools across different abstraction layers and matching their traces. Hence,

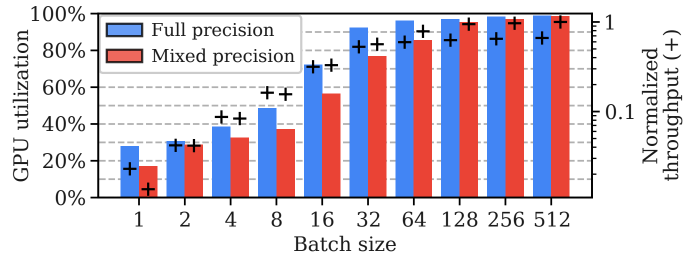

Fig. 1: GPU utilization and throughput when executing ResNet-50 training with different batch sizes.

design decisions are often based either on high-level metrics (e.g., GPU utilization), which can be misleading because they may not reflect the utilization of the internal components of the GPU architecture; or, on low-level metrics (i.e., tensor cores utilization), which cannot capture the efficiency of the host/device interactions happening at the high level.

As an illustration, let's consider one training loop of ResNet-50 running on a single-GPU system. We can define GPU utilization as the proportion of time the GPU is actively used during training time. One could argue that a higher GPU utilization is desirable, as it signifies that the training process can consistently harness the superior GPU computing power. In practice, however, certain training processes can attain shorter training times by compromising GPU utilization. Using mixed-precision in our example, we observe a decrease in the overall GPU utilization while also increasing the throughput for the most performant batch sizes (*see Fig. 1*).

Our **main goal** is to evaluate how efficiently ML training workloads use the computing resources of modern GPUs. To this end, we make the following contributions:

- We describe an ideal reference execution of a GPU-accelerated ML training loop and identify relevant metrics that can be measured using existing profiling tools.
- We produce a coherent integration of the traces obtained from each profiling tool and explain how to evaluate utilization at four abstraction levels of GPU resources.
- We analyze the impact of different software optimizations (e.g., mixed-precision, various ML frameworks, and execution modes) on the throughput and the associated utilization at multiple levels (i.e., GPU, SM subpartitions, issue slots, and tensor cores) in two modern GPUs running representative ML training workloads.

The source code of our tools and traces is freely available at https://gite.lirmm.fr/adac/delestrac2024multilevel.

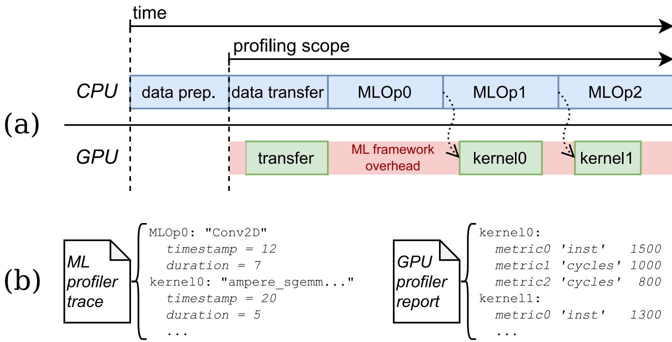

Fig. 2: (a) Typical eager execution trace. (b) CPU and GPU events are listed in the ML profiler trace file. GPU's performance counters are listed in the GPU profiler report.

II. Related works

NVIDIA provides several profiling interfaces [8]–[10] for developers to evaluate GPU performance bottlenecks. ML frameworks also provide framework-specific profiling tools for developers to evaluate the performance of their models [11], [12]. In this paper, we gather metrics using both NVIDIA Nsight Compute [10] and ML frameworks profilers [11], [12].

Previous works have also leveraged these existing profiling tools to provide analyses on ML workloads. Some directly use these tools to analyze ML workloads [4]. Others develop meta-tools that can identify bottlenecks by doing some automatic post-processing of the data reported by one profiler [13]–[15], or aggregated data from multiple profilers [16], [17]. While most of these related works evaluate performance through metrics like throughput and latency, some works also evaluate GPU utilization [4], [16], [17]. However, to the best of our knowledge, this is the first work to systematically evaluate GPU resource utilization at multiple levels and across multiple training workloads, ML frameworks, and execution modes.

III. Efficient ML training loop execution on GPU

In this section, we describe an ideal reference ML training loop execution on GPU at multiple hardware abstraction levels. For each level, we identify metrics that can be measured to evaluate the utilization of the GPU computing resources.

GPU level. *GPU utilization* is the ratio of kernel time over the total execution time. The time during which the GPU is inactive is the *ML framework overhead*. At runtime, for each GPU-compatible operation that has to execute, the ML framework has to launch one or multiple kernels on the GPU. Before launching these kernels, the ML framework needs to execute other enabling tasks on CPU, such as parsing the Python code, copying data between CPU and GPU, etc. Ideally, this setup time would be hidden by the parallel execution of previously launched GPU kernels. However, in practice, the setup time creates bubbles of GPU inactivity (*see Fig. 2a*).

GPU utilization can increase either by extending kernel execution time while keeping the same ML framework overhead, or by reducing the ML framework overhead while keeping the kernel execution time. In practice, a common and easy way to extend the kernel execution time without changing the model is to increase the batch size. Reducing the ML framework's overhead can be done using more optimized modes of execution than eager, like just-in-time (JIT) compilation. These optimized modes leave the flexibility of Python [18] to enable high-level code transformations or optimizations across multiple operations of the model (e.g., dead code elimination, constant folding, arithmetic simplification, kernel fusion).

However, GPU utilization alone can be misleading as it only shows how well the setup time is hidden by the execution of GPU kernels. *Throughput*, defined as the number of input samples processed per second, is a better metric to compare different workloads, but it lacks specificity in pinpointing performance bottlenecks.

SM level. When a GPU kernel is launched, its threads are grouped in thread blocks and automatically distributed across multiple streaming multiprocessors (SMs) by the GPU scheduler. The scheduler keeps SMs busy by distributing the thread blocks evenly. However, during the beginning and end of the execution, some SMs may run out of thread blocks to execute. This can lead to SM underutilization [19]. At a given cycle, we consider an SM as active if it has at least one warp (i.e., a set of 32 threads executing the same instruction) to execute. SMs of modern NVIDIA GPUs are composed of four subpartitions. For finer granularity, we choose to evaluate utilization at SM subpartition (SMSP) level. We define *SMSP active utilization* as the number of active SMSP cycles over the total number of elapsed cycles on the GPU. Hence, the difference between GPU and SMSP utilization evaluates the GPU scheduler's ability to effectively parallelize threads across SMSPs.

Instruction issue level. An active SMSP does not necessarily imply that warp instructions are being issued. Indeed, the warp scheduler (internal to the SMSP) can stall the issuing of a warp instruction for different reasons (e.g., waiting for data to be fetched from memory, or waiting for a dependent instruction to finish execution). A cycle during which the warp scheduler is stalled (i.e., cannot issue any instruction to the GPU cores) is called *stall cycle*. Otherwise, the cycle is called *issue cycle*. We define the *SMSP issue slot utilization* as the ratio of issue cycles over the total number of elapsed cycles on the SMSP. In modern GPUs, each SMSP can issue one warp instruction per active cycle. Hence, the SMSP active utilization is an upper bound for the SMSP issue slot utilization.

Tensor core level. Tensor cores (TC) are specialized GPU cores designed to accelerate matrix multiplications. As a result, peak GPU performance in terms of Floating Point Operations Per Second (FLOPS) can only be achieved through the use of tensor cores. Ideally, maximizing the *tensor core utilization* relies on three conditions. First, the number of TC instructions should dominate the total number of issued instructions (i.e., SMSP issue slot utilization). Second, the number of TC active cycles should dominate the total number of active SMSP cycles (i.e., SMSP active utilization). Finally, the issued TC instructions should achieve peak throughput of the tensor cores. Due to the latency and instruction bandwidth of the tensor cores, this last condition is frequently not met [5].

IV. Profiling methodology

ML training workloads comprise thousands of identical iterations. Instead of profiling the complete training, we limit our scope to a single iteration and ignore the data preprocessing steps that are executed on CPU (*see Fig. 2a*). We choose the target iteration using a preliminary run of multiple iterations. We select the first iteration with a stable execution time and ignore the iterations including initialization overheads. These overheads are negligible in comparison with the overall training time and do not reflect the characteristics of an average training iteration.

We gather metrics from ML framework profilers and the GPU kernel profiler. The ML framework profilers (i.e., TensorFlow Profiler [11] or PyTorch Profiler [12]) provide the list of events as JSON files that we parse to gather our metrics of interest. We list the selected high-level metrics at the top of TABLE I. Instead, the GPU kernel profiler (i.e., NVIDIA Nsight Compute [10]) provides a report file that lists all the executed kernels during the profiled iteration along with the values of the GPU performance counters. We list the selected low-level metrics at the bottom of TABLE I.

Note that the tensor core metrics that we gather only cover two of the three conditions required to achieve peak tensor core utilization (described in Section IV). This limitation is because the GPU profiler only provides the total number of instructions that have been issued to the tensor cores, without distinguishing between their different types. Hence, we evaluate tensor core utilization between two bounds: (1) the ratio of tensor core instructions over the amount of issued instructions, and (2) the ratio of active tensor core cycles over the amount of active SMSP cycles.

Analyzing the profiled metrics at the granularity of ML model architectures (e.g., per DL layer type) is not straightforward. On the one hand, the ML framework profiler does not gather the low-level metrics from the GPU performance counters for each kernel. On the other hand, the GPU kernel profiler cannot provide any information regarding the ML framework runtime. Hence, we need to identify the correspondence between kernels from the ML profiler trace and kernels from the GPU profiler report.

Previous works have done this by running a tracer in parallel with the ML profiler [16], allowing the analysis code to run in parallel with the profiling of the ML workload. Instead, we propose to match the traces offline by comparing kernel metrics that are common between both profiling tools (e.g., kernel name, block and thread dimensions, kernel duration, memory usage, etc.). Based on these metrics, we establish a correlation matrix between the kernels from the ML profiler trace and the kernels from the GPU profiler report. We then use this correlation matrix to identify the corresponding kernels and match both high-level and low-level metrics.

TABLE I: High-level (top) and low-level (bottom) metrics

Name	Description
Total execution time	Total duration of the traced iteration
GPU kernel exec. time	Aggregate duration of all the GPU events
GPU utilization	GPU kernel exec. time / Total execution time
Achieved throughput	Batch size / Total execution time
SMSP elapsed cycles	Elapsed GPU cycles counted at SMSP level
SMSP active cycles	SMSP cycles with at least 1 warp active
SMSP active utilization	SMSP active cycles / SMSP elapsed cycles
SMSP issue cycles	SMSP cycles with an instruction issued
SMSP issue slot utilization	SMSP issue cycles / SMSP elapsed cycles
TC active cycles	SMSP cycles with at least 1 active tensor core
TC active utilization	TC active cycles / SMSP elapsed cycles
TC issue cycles	Number of tensor core instructions issued
TC issue slot utilization	TC issue cycles / SMSP elapsed cycles

TABLE II: Main GPU features

	NVIDIA A100	NVIDIA V100
Architecture	Ampere	Volta
SMSPs	432	320
DRAM Memory	80 GiB	32 GiB
Tensor Cores (Peak TFLOPS)	432 (312)	640 (125)

V. Experimental setup

Workloads. We choose three well-known supervised DL workloads (i.e., ResNet-50, BERT, and DLRM) from the MLPerf training benchmark suite [20]. These models are representative of common model architectures like CNN, Transformer-based, and recommendation models, respectively. We use PyTorch (PT) and TensorFlow (TF) as the reference ML frameworks, due to their maturity. ResNet-50 (PT), BERT (PT and TF), and DLRM (TF) implementations are downloaded from the NVIDIA repository [21]. DLRM (PT) implementation is downloaded from the MLPerf Training benchmarks reference models. ResNet-50 (TF) implementation is provided directly inside the TensorFlow library [22]. All of the training parameters are chosen following the MLPerf Training benchmarks rules. Our results show less than 1% standard deviation across three runs for each workload.

ML frameworks. We use PyTorch and TensorFlow with two different modes of execution: eager execution and just-in-time (JIT) compilation. For eager execution, we use the default eager execution runtimes. For JIT compilation, we use XLA JIT for TensorFlow and the TorchScript JIT backend of PyTorch. Additionally, we run each mode using both full-precision (FP) and mixed-precision (AMP).

GPUs. We use NVIDIA V100 and A100 GPUs paired with an AMD Milan EPYC 7543 CPU with 32 cores at 2.8GHz. TABLE II lists the main GPU features.

Profiling tools. We use TensorFlow Profiler [11] and PyTorch Profiler [12] to gather high-level metrics and NVIDIA Nsight Compute [10] to gather low-level kernel metrics.

VI. Results

In this section, we measure and analyze the GPUs' training compute efficiency from two perspectives: *(A) performance vs. memory utilization and (B) compute resource utilization*. Additionally, we draw some general implications and insights.

A. *Performance vs. Memory Utilization*

GPU memory & Batch size. We evaluate the raw performance by measuring training throughput (i.e., amount of processed items per second). Industry and research-leading benchmarks such as MLPerf [20] rank the participants on training time, which is inversely proportional to throughput. Fig. 3 shows the normalized throughput (y-axis) of running ResNet-50 with various batch sizes (annotated on the markers) and ML framework options, including the corresponding GPU memory allocated (x-axis), using both the A100 (Fig. 3a) and V100 (Fig. 3b) GPUs. Only batch sizes that use more than 10% of the GPU memory are shown. We include this workload as an illustration, but the other workloads follow a similar trend.

We can observe that both GPU memory allocated and throughput increase with growing batch size. For example, using the A100, transitioning from batches of 128 to 2048 images, TF XLA JIT with mixed-precision achieves a $3.8\times$ increase in throughput, while allocating around $15\times$ more GPU memory. However, while the GPU memory allocated grows linearly with the batch size, throughput saturates when approaching the biggest batch sizes. For the same example, going from 1024 to 2048 only increases throughput by 3% (1% for FP), while doubling ($1.98\times$) the allocated memory.

This saturation is less visible when using the V100, which has less memory capacity than the A100 (i.e., 32GB vs. 80GB, respectively). In fact, TF XLA JIT still achieves a 58% (40%) increase in throughput when going from batches of 128 (256) to 256 (512) images (i.e., biggest batch size that can fit in the V100's memory) using full-precision (mixed-precision).

Key takeaway 1. *The A100 provides enough memory to saturate throughput, eliminating utilization gaps caused by limited memory capacity and the push for larger batch sizes.*

ML framework execution modes. We compare memory usage and throughput across eager execution and JIT compilation for both TF and PT. Fig. 3 shows different markers for eager execution (round markers) and JIT compilation (square markers). We make two observations. First, TF XLA JIT allocates less GPU memory than TF eager for the same batch size, which increases throughput (as discussed in our first

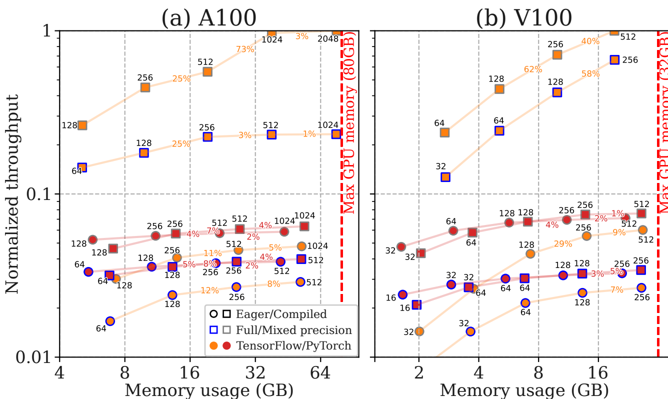

Fig. 3: Normalized throughput vs. GPU memory usage for ResNet-50 workloads on (a) A100 and (b) V100 GPUs.

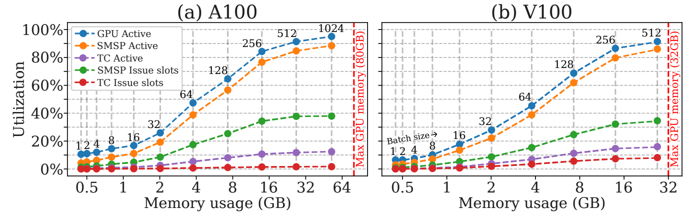

Fig. 4: Multi-level utilization vs. GPU memory usage. TensorFlow eager execution of a mixed-precision ResNet-50 across various batch sizes on (a) A100 and (b) V100.

takeaway). For example, running a TF eager execution requires 65% of the A100's memory for a batch size of 512, whereas XLA JIT only requires 48% of the memory for the same batch size. As a result, XLA JIT can fit larger batch sizes within the GPU memory. Second, XLA JIT can achieve $8\times$ ($25\times$) higher throughput than TF eager execution in the A100 (V100).

When comparing precision modes, we can make two observations. First, mixed-precision (grey markers) consistently uses around 50% of the memory allocated by full-precision (blue markers). This enables to fit larger batch sizes in the same amount of memory, which increases throughput compared to full-precision. Additionally, mixed-precision provides a boost in throughput for the same respective batch sizes as full-precision. For example, using PT, mixed-precision achieves around $1.5\times$ ($2\times$) the throughput of full-precision for the same batch sizes with the A100 (V100).

Key takeaway 2. *JIT compilation and mixed-precision increase throughput by fitting larger batch sizes in GPU memory.*

When comparing ML frameworks, we observe that PT eager uses less memory and achieves better throughput than TF eager, particularly with large batch sizes. For example, using full-precision with a batch size of 512, PT eager achieves $1.3\times$ the throughput of TF eager and uses 12% less memory. However, when using JIT compilation, TF XLA JIT outperforms every other mode of execution, achieving the best throughput across all batch sizes and using less memory. For example, using full-precision with a batch size of 512, TF XLA JIT achieves $5.8\times$ the throughput of PT JIT and uses 17% less memory.

Key takeaway 3. *Generally, TF XLA JIT significantly outperforms PT TorchScript JIT in both throughput and memory.*

B. *GPU Compute Resource Utilization*

We evaluate the utilization of the GPU's compute resources as GPU utilization, SM subpartition utilization, issue slot utilization, and tensor core utilization, as described in Section IV. Fig. 4 shows the GPU resource utilization at each level for a ResNet-50 TF eager mixed-precision run on the A100 and V100. We use this run as an illustration of the measured utilization at different levels of the GPU hardware abstraction for different batch sizes. Instead, Fig. 5 shows the utilization results for all the workloads but only for the batch size achieving the highest throughput in the A100 GPU.

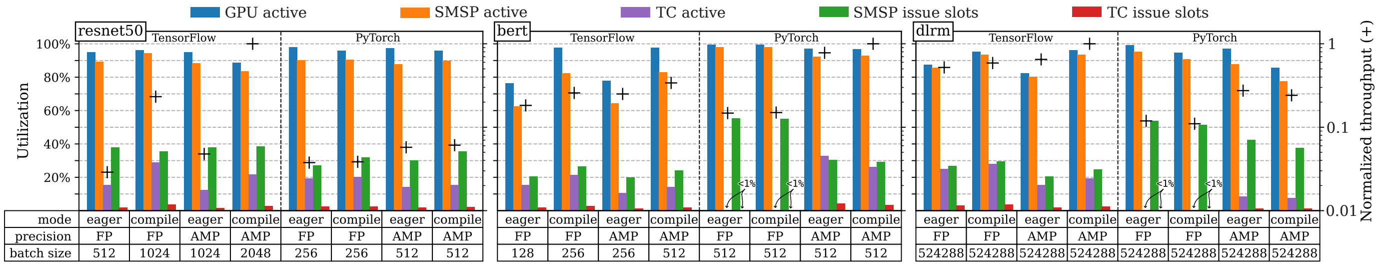

Fig. 5: Multi-level utilization (A100 workloads): GPU, SMSP (active & issue slots), and tensor core (active & issue slots). Batch size chosen based on best throughput value for each workload. Throughput is normalized by best value for each model.

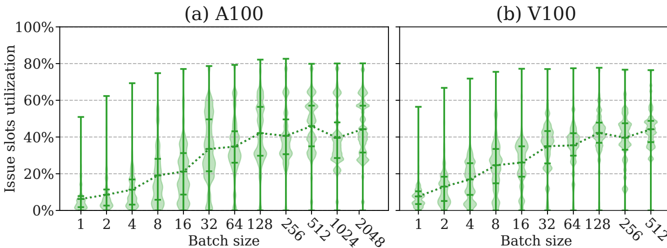

Fig. 6: Distribution of SMSP issue slot utilization for all kernels of ResNet-50 using XLA JIT with mixed-precision for multiple batch sizes on (a) A100 and (b) V100 GPUs.

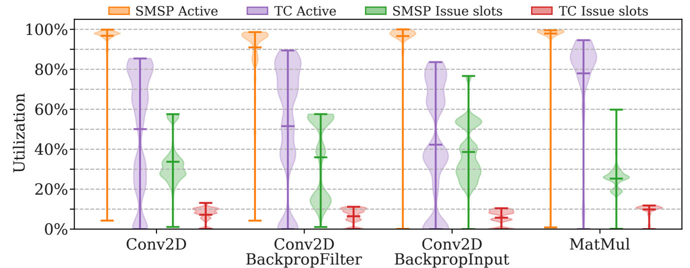

Fig. 7: Per-layer SMSP and TC utilization distributions (showing only layers with over 3% average TC issue slot utilization).

GPU utilization. In Fig. 4, we observe that GPU utilization increases as the batch size increases, reaching a maximum of 95% and 91% for the A100 and V100, respectively. We observed this link between the increase of batch size and the increase of GPU utilization for all of our workloads. In Fig. 5, we can observe that BERT using TF eager execution achieves the lowest GPU utilization, which tops at 76% using full-precision and 78% using mixed-precision. We also observe that PT achieves a slightly (2 to 6%) higher GPU utilization than TF. However, this higher GPU utilization does not always lead to a higher throughput. For example, DLRM full-precision achieves 99.3% GPU utilization with PT eager compared to 87.4% with TF eager. However, TF eager achieves $4.4\times$ higher throughput than PT eager, which can likely be attributed to the higher utilization of tensor cores.

Key takeaway 4. *Higher GPU utilization does not correlate with higher throughput.*

SMSP active utilization. Fig. 4 shows that SMSP active utilization follows the same trend as GPU utilization across different batch sizes for ResNet-50 TF eager with full-precision. We observed the same behavior for all our workloads. However, as discussed in Section IV, GPU utilization is an upper bound for SMSP active utilization. We observe that SMSP active utilization is consistently 1% to 15% lower than GPU utilization. Similarly to GPU utilization, we observe no correlation between SMSP active utilization and achieved throughput across workloads.

Key takeaway 5. *SMSP active utilization mirrors GPU utilization across batch sizes, staying 1% to 15% lower.*

SMSP issue slot utilization. Fig. 4 shows that SMSP issue slot utilization follows the same trend as GPU and SMSP active utilization. We observed the same behavior for all the workloads. However, Fig. 5 shows that average SMSP issue slot utilization generally stays below 40%, except for BERT and DLRM using PT full-precision, where it approaches 54%.

To better understand SMSP issue slot utilization, Fig. 6 shows the distribution for all kernels of the ResNet-50 model using mixed-precision XLA JIT for multiple batch sizes for the A100 (Fig. 6a) and V100 (Fig. 6b) GPUs. Average SMSP issue slot utilization (dotted line on Fig. 6) follows the same trend as seen in Fig. 4, and saturates near 40%. However, data distribution (violin plots and vertical bars in Fig. 6) shows maximum SMSP issue slot utilization ranging between 50% and 80%, generally increasing with batch size. We also observe that although the average (and maximum) value is very similar for both GPUs, the A100 includes a higher population of kernels with an SMSP issue slot utilization above 60%.

Key takeaway 6. *The average SMSP issue slot utilization increases with the batch size, but it seldom exceeds 40%.*

Tensor core utilization. We compare tensor core utilization across all workloads for different batch sizes using two metrics: TC active utilization and TC issue slot utilization. Fig. 4 shows an increase in both metrics as the batch size increases for ResNet-50 using TF eager execution with full-precision. However, Fig. 5 shows that tensor cores are active for less than 35% of the total execution time. Furthermore, less than 5% of the total execution cycles are spent issuing TC instructions.

To gain more insights, Fig. 7 shows the tensor core utilization distribution for all the kernels in the TF eager workloads, aggregated by layer. We observe that only MatMul-based layer

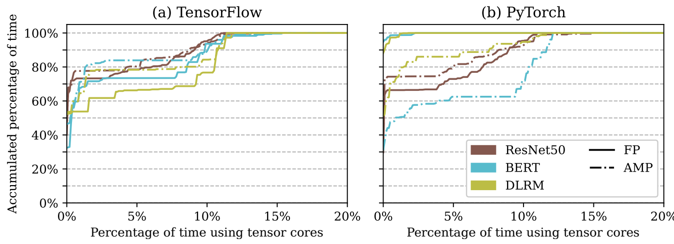

Fig. 8: Cumulative distribution of kernels' TC issue slot utilization for eager workloads using the A100 GPU. Batch size chosen based on best throughput value for each workload.

types, namely convolution (i.e., Conv2D) and fully connected (i.e., MatMul) layers, achieve TC issue slot utilization above 3%. On the one hand, the average TC active utilization can reach close to 80% for pure MatMul operations, while it is limited to around 50% for convolution layers. On the other hand, the average TC issue slot utilization tops at 10%, with some kernels achieving near 15% at peak. These same layers all achieve above 90% average SMSP active utilization and up to 40% average SMSP issue slot utilization.

To better understand how this relatively low issue rate of tensor core instructions impacts performance, Fig. 8 shows the cumulative training time as a function of TC issue slot utilization, for the eager workloads on TensorFlow (a) and PyTorch (b). The kernels that do not issue any tensor core instructions amount to 30 to 95% of the total A100 training time depending on the workload. Furthermore, kernels with less than 12% of TC issue slot utilization represent more than 99% of the total GPU kernel execution time of all workloads. Despite these low utilization rates, the use of tensor cores significantly increases training throughput (see Fig. 5). However, we observe that even the workloads benefiting the most from tensor cores are now limited by the execution time of kernels that do not use tensor cores (Amdahl's law at play).

Key takeaway 7. *The majority of the kernels do not use tensor cores. Kernels that use tensor cores more extensively amount to a small proportion of the total GPU execution time.*

C. Implications and Insights

To achieve sustainable performance improvements within ML training workloads, it is important to maintain a balanced utilization of key architectural resources. GPUs have increased their memory capacity in recent generations, enabling enhanced throughput and higher utilization through the support of larger batch sizes. However, our experiments on representative workloads suggest a plateau has been reached, and the additional memory in the A100 no longer leads to enhanced utilization ratios. We also show that modern GPUs can achieve impressive acceleration but typically operate below 50% of their instruction-issuing potential. Furthermore, we observe that the tensor cores, which are the instructions delivering the highest raw computational power, are kept idle most of the time, and the evaluated ML training workloads are now constrained by kernels not using tensor cores. Thus, our results suggest that the current GPU paradigm is reaching a saturation point, and motivate further research into programmable architectures to sustainably accelerate ML training workloads.

VII. Conclusion

In this work, we analyze the efficiency of executing ML training workloads on modern GPUs by describing an ideal GPU-accelerated ML training loop and identifying relevant performance metrics. We combine traces from existing profiling tools and compare the execution of various ML training workloads to the ideal loop. We present our results with seven key takeaways, showing that high utilization is typically achieved at the GPU level. Yet, the average instruction issue slot utilization remains below 50%, with tensor core instructions reaching less than 5.2%. We believe this work highlights the need for advanced profiling to unravel GPU limitations.

VIII. Acknowledgement

This work was performed using HPC/AI resources from GENCI-IDRIS (Grant AD011012967).

References

[1] T. Brown *et al.*, "Language models are few-shot learners," in *Proceedings of NeurIPS*, 2020.
[2] A. Reuther *et al.*, "AI and ML accelerator survey and trends," in *Proceedings of HPEC*, 2022.
[3] C. Yang *et al.*, "Hierarchical roofline analysis for GPUs," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 20, 2020.
[4] S. Verma *et al.*, "Demystifying the MLPerf training benchmark suite," in *Proceedings of ISPASS*, 2020.
[5] W. Sun *et al.*, "Dissecting tensor cores via microbenchmarks: Latency, throughput and numeric behaviors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, 2022.
[6] P. Delestrac *et al.*, "Demystifying the TensorFlow eager execution of deep learning inference on a CPU-GPU tandem," in *Proceedings of DSD*, 2022.
[7] S. Chetlur *et al.*, "cuDNN: Efficient primitives for deep learning," 2014.
[8] "NVIDIA CUDA Profiling Tools Interface (CUPTI)." [Online]. Available: https://developer.nvidia.com/cupti
[9] "NVIDIA Management Library (NVML)." [Online]. Available: https://developer.nvidia.com/nvidia-management-library-nvml
[10] "NVIDIA Nsight Compute." [Online]. Available: https://developer.nvidia.com/nsight-compute
[11] "TensorFlow profiler: Profile model performance." [Online]. Available: https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras
[12] "PyTorch profiler." [Online]. Available: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html
[13] K. Zhou *et al.*, "GPA: A GPU performance advisor based on instruction sampling," in *Proceedings of CGO*, 2021.
[14] A. Saiz *et al.*, "Top-down performance profiling on NVIDIA's GPUs," in *Proceedings of IPDPS*, 2022.
[15] H. Zhang *et al.*, "Understanding the performance of GPGPU applications from a data-centric view," in *Proceedings of ProTools*, 2019.
[16] C. Li *et al.*, "XSP: Across-stack profiling and analysis of machine learning models on GPUs," in *Proceedings of IPDPS*, 2020.
[17] J. Gleeson *et al.*, "RL-Scope: Cross-stack profiling for deep reinforcement learning workloads," in *Proceedings of MLSys*, 2021.
[18] H. He, "Making deep learning go brrrr from first principles," 2022. [Online]. Available: https://horace.io/brrr_intro.html
[19] S. Park *et al.*, "Analysis of thread block scheduling algorithms for general purpose GPU systems," in *Proceedings of CSDE*, 2021.
[20] P. Mattson *et al.*, "MLPerf training benchmark," in *Proceedings of MLSys*, 2020.
[21] NVIDIA, "NVIDIA deep learning examples." [Online]. Available: https://github.com/NVIDIA/DeepLearningExamples
[22] H. Yu *et al.*, "TensorFlow Model Garden," 2020. [Online]. Available: https://github.com/tensorflow/models